Arrive Technologies Inc.

Guidelines

ATVN

Verilog Coding Standards

Abstract: This document describes the Verilog Programming Style to be used in the logic development. A Set of conventions is described covering various constructs of the language. These conventions are intended to provide a high level of uniformity in the code produced by different programmers thus achieving readability and maintainability.

Revision: ***3.1***

Last Updated: ***03-Aug-06***

Author:

- AT ASIC Design Group

Document History

Revision	Date	Description
1.0	May 2001	Initial Version
2.0	Jan 2003	Changed for new RED-BLUE chips
3.0	July 2003	After a review on July-29-03
3.1	Aug 2006	New Template updated

Contents

Figures

1. Abstract

The purpose of this document is to define one style of programming in Verilog. All non-optional rules and recommendations provided in this document are followed when developing Arrive Technologies logic level. These rules and recommendations are intended to encourage higher quality code in terms of uniformity, readability, robustness, reliability and maintainability. It also allows programmers to work on code written by others with fewer overheads in adjusting to stylistic differences.

2. Acronyms

Empty section.

3. Related Documents

Empty section.

4. Requirements

Empty section.

5. Revision Changes

Revision	Changes
3.1	1. New Template Updated
3.0	1. Changed source code header (6.1.1) 2. Changed the way to name inputs and outputs: removing prefix i for input, prefix o for output (6.2.3). 3. Changed the way to name wires and registers: removing prefix w for wire, prefix r for register (6.2.4). 4. Changed the way to name active low signal: adding suffix _ (underscore) for active low signal (6.2.5). 5. Removed the Style 1 in Project Hierarchy section (6.5) 6. Changed directory TEST to RSIM in Project Hierarchy section (6.5) 7. Added RTL Mandatory section (6.4)
2.0	1. Added Revision Changes section (5.). 2. Changed source code file layout and header (6.1.1). 3. Changed the way to name inputs and outputs: adding prefix i for input, prefix o for output (6.2.3). 4. Changed the way to name wires and registers: adding prefix w for wire, prefix r for register (6.2.4). 5. Changed the way to name active low signal: adding suffix x for active low signal (6.2.5). 6. Added testcase files (6.1.3). 7. Added the Style 2 in Project Hierarchy section (6.4.2). 8. Minor corrections

6. Verilog Programming Conventions

6.1. Layout Conventions

6.1.1. Source File Layout

Every file containing Verilog source code, denoted by **.v** extension, must contain a standard file heading in the beginning followed by the module body (i.e. a set of functions). The file heading consists of the blocks described below. Note that blocks are separated by one or two blank lines.

- **Copyright:** This consists of a single-line Verilog comment containing the appropriate copyright information.
- **Title:** The title consists of a one-line Verilog comment indicating the file name followed by a short description. The name must be the same as the file name.
- **Description:** This consists of a paragraph of Verilog comment (with a proper border as shown below) summarizing the purpose of the source file.
- **Author:** This author consists of a one-line Verilog comment indicating author's e-mail address.
- **Created:** This consists of a one-line Verilog comment indicating the first day when source file was created.

- **History:** This consists of a one-line Verilog comment indicating the last day when source file have changed, by whom and why.

The format of these blocks is shown in the following example:

```
////////////////////////////////////////////////////////////////////////////////<-80 chars
//
//  Arrive Technologies
//
// Filename         : ds_sm.v (Downstream Data Path State Machine)
// Description      : This module implements the downstream data path state machine
//
// Author           : authorname@hostname.com.vn
// Created On       : Mon Jul 28 19:29:44 2002
// History (Date, Changed By)
//              Jul 31 2002, authorname, added framesyn.
//              Jul 30 2002, authorname, fixed downstream counter.
//
//
////////////////////////////////////////////////////////////////////////////////<-80 chars

module ds_sm
    (
    rst_,
    clk38,
    clk19,

    signal1,    // connects to Module 1
    signal2,
    signal3,

    signal7,    // connects to Module 2,
                // synchronized with clk19
    signal8     // connects to Module 3
    );

////////////////////////////////////////////////////////////////////////////////<-80 chars
// Port declarations

input           rst_;                   // underscore suffix for active low signal
input           clk38;
input           clk19;

input   [6:0]   signal1;
output  [12:0]  signal2;
input   [7:0]   signal3;

input           signal7;

output          signal8;

////////////////////////////////////////////////////////////////////////////////<-80 chars
// Output declarations

reg             [12:0]  signal2;

////////////////////////////////////////////////////////////////////////////////<-80 chars
// Parameter declarations

parameter       PARA1   = 2'b11; // using uppercase for parameter names

////////////////////////////////////////////////////////////////////////////////<-80 chars
// Local signal declarations


////////////////////////////////////////////////////////////////////////////////<-80 chars
// Logic instantiation
```

6.1.2. Testbench File Layout

Testbench files, denoted by a **.vt** extension, must contain a standard file heading as Source File heading.

6.1.3. Testcase File Layout

Testcase files, denoted by a **.vc** extension, must contain a standard file heading as Source File heading.

6.1.4. Function Layout

Every function must be preceded by a function heading with a proper border of Verilog comment. The heading consists of the blocks described below. The blocks are separated by single blank lines and may contain blank lines within the block.

- **Title:** One line containing the function name followed by a short description. The name in the title must be the same as the declared name.
- **Description:** One or more paragraph(s) describing in detail the functionality implemented by the function.

Immediately following the function heading should be the function declaration. The format of these blocks is shown in the following example:

```
////////////////////////////////////////////////////////////////////////////////<-80 chars
// Comments for the following logic

wire   signal1;
assign signal1 = ^signal2;

reg [12:0]     signal3;
always @( posedge clk19 or negedge rst_ )
    begin
    if (!rst_)
        begin
        signal3 <= 13'b0;
        end
    else
        begin
        if (|signal3)
            begin
            signal3[2] <= signal4;
            end
             :
             :
        end
    end
```

6.1.5. Code Layout

This section describes the conventions for the graphic layout of Verilog code. The main goal behind the following recommendations is readability. Programmers are encouraged not to use multi-line cryptic expressions and statements.

6.1.5.1. Vertical Spacing

- Use blank lines to make code more readable and to group logically related sections of code together. Put one blank line before and after comment lines.
- Do not use more than 2 blank lines for vertical separation.
- Do not put more than one declaration on a line. Each variable and function argument must be declared on a separate line. Do not use comma-separated lists to declare multiple identifiers.
- Do not put more than one statement on a line. The only exception is the for statement, where the initial, conditional, and loop statements may be written on a single line:

```
for (reg = 0; reg < count; reg = req + 1)
    begin
    end
```

The **if** statement which has a simple structure is also an exception, as shown below:

```
if (reg > count) reg <= count;
```

6.1.5.2. Horizontal Spacing

- All lines should display in 80 columns, with tab size set to 4.
- Put spaces around binary operators and after commas. Do not put spaces before and after open brackets and parenthesis.

For example:

```
modulename instantiatename (.signal1(signal1),  .signal2(signal2));
```

-or-

```
modulename instantiatename
    (
    .signal1(signal1),
    .signal2(signal2)
    );
```

- Continuation lines should line up with the part of the preceding line they continue:

```
reg_ex = (val1 | val2) &
         (val3 | val4);

if ((reg == 8'hfa) &&
     val1 == 2'b01))
    begin
    end

assign wire1 = (code == 2'b01) ? (true_expression_1) :
               (code == 2'b10) ? (true_expression_2) :
               (code == 2'b11) ? (true_expression_3) : (false_expression);
```

6.1.5.3. Indentation

- Indentation levels are every four characters (i.e. columns 1, 5, 9, 13, etc.).
- All the layout headings described earlier and function declarations start in column one.
- The **else** of a conditional has the same indentation as the corresponding **if**. Thus the form of the conditional is:

```
if (condition) statement          // in case of there is one statement only.
else statement
```

-or-

```
if (condition)                                  // in case of there are many statements.
    begin
    statement(s)
    statement(s)
    end
else
    begin
    statement(s)
    statement(s)
    end
```

The form of the conditional statement with an **else if** is:

```
if (condition)
    begin
    statement(s)
    end
else if (Condition)
    begin
    statement(s)
    end
else
    begin
    statement(s)
    end
```

- The format for the **case** statement is:

```
case (signal3)
    pvalue1:
       begin
       end
    pvalue2,
    pvalue3,
    pvalue4:
       begin
       end
    default:
       begin
       end
endcase
```

- The format for the **always** and **initial** statements is:

```
always @( posedge clk19 or negedge rst_ )
    begin
    if (!rst_)
        begin
        end
    else
        begin
        if ()
            begin
            end
        end
    end

initial
    begin
    ...
    end
```

- The format for module instantiation is:

```
core    icore   // in case of instantiating a user-defined module
    (           // or a long-list macro
    .signal1(signal1),
    .signal2(signal2),
    .signal3(signal3)
    );
```

-or-

```
macro   imacro (signal1, signal2, signal3);   // it is acceptable if instantiating
                                              // a short-list macro
```

- Comments must have the same indentation level as the section of code to which they refer.
- Begin and End have the same indentation as the code they enclose.

6.1.5.4. Comments

- Comments within the code should precede the section of code to which they refer and have the same level of indentation. In most cases, they should be separated from the code by single blank line.

1. Single-line comments should begin with the open-comment and end with the close-comment as shown below:

```
// This is the correct format for a single-line comment
assign code = pvalue;
```

-or-

```
assign code = pvalue;  // This is the correct format for a single-line comment
```

2. Multiline comments should begin and end with the open-comment and close-comment on separate lines. Some examples are shown below. The same format should be used for all multiline comments within a module.

```
/*
 This is one of the correct formats for a multiline comment
 in a section of code.
*/
assign code = pvalue;
```

It is also acceptable to line up each part of the multiline comment as shown.

```
// This is the correct format for a multiline
// comment in a section of code

/*
This is the correct format for a multiline comment
a section of code.
*/
assign code = pvalue;
```

It is even acceptable to do either of the following:

```
/*
        It is okay to indent the comment
        From the open-comment and close-comment
*/

/*
 * It is acceptable to line up all the stars
 * as such
 */
```

6.2. Naming Conventions

- When creating names, remember that the code will be written once or twice, but read many times. Make names meaningful and readable, and avoid obscure abbreviations. In general, the following guidelines must be followed.
- In general, names should be **lowercase** and be composed of words, abbreviations, and acronyms combined together with **underscores (as rarely as possible)**. The composed word should form a close description of the object named. The name should be long enough for the reader to be able to determine what the object is for. Besides, the name should be shorter than 10 characters for good synthesis procedure.
- Minimize the use of abbreviations.
- Use abbreviations consistently.

For example:

```
input             prs;          // it means processor write strobe
input      [4:0]  framecnt;      // it means frame count, do not use frame_count
                                 // because frame_count is so long.
input             t_run;        // it means time run, do not use trun
                                 // because trun is easily misunderstanding .
```

6.2.1. Source, Testbench, and Testcase File Names

The prevailing theory on naming is that the name be *meaningful*. Commonly this means that each module name in a subsystem has a short prefix (two to five characters) which implies the subsystem. Only lower case letters should be used when naming source and header files. The file name should be exactly the same as the module name in the file.

For example:

```
tcmain.v - Telephony Controller main module

tcprov.v - Telephony Controller provisioning module
```

```
tctb.vt – Telephony Controller TestBench
tctb.vc – Telephony Controller TestCase
```

6.2.2. Reset Signal Name

Names of reset signal must contain a string **rst** and must be **active low**.

For example:

```
input      rst_;
```

6.2.3. Clock Tree Signal Name

Names of clock tree signal must contain a string **clk**.

For example:

```
input      sysclk;
```

6.2.4. Constant Names

Names of constant (defined via **`define** or **parameter**) must be **uppercase**.

For example:

```
`define        CMAX    8'd5
parameter      CMIN =  4'd1;
```

6.2.5. Input and Output Names

- Names of input and output must have a comment, which indicates this signal changed at which clock domain, positive edge or negative edge, static or synchronous, active high or active low.
- Default thought is that input and output signals are synchronized with a main input clock. If synchronized with other input clocks, comments must be described.
- Input and output signals should be **active high**.

For example:

```
input   signal1name;   // static
output  signal2name;   // @-clk38 (changed at negative edge of clock 38)
```

6.2.6. Wire and Register Names

- Names of wire and register must have a comment, which indicates this signal changed at which clock domain, positive edge or negative edge, static or synchronous, active high or active low.
- Wires and registers should be active high.

For example:

```
wire    signal1name;   // static
reg     signal2name;   // @-clk38 (changed at negative edge of clock 38)
wire    signal3name_;  // active low
```

6.2.7. Active Low Variable Names

Active low variable names must have an _ (underscore) suffix respectively. For example:

```
wire    activelow_;
```

6.3. Complex Conditionals

- Complex conditional statements can be cryptic and hard to comprehend but inevitable sometimes. Proper indentation can improve this difficult situation quite a bit as shown below.

- before -

```
if  ((((signal1 == 2'b10) &&
    (signal2 != c_value1)) ||
    (signal3 == 2'b10)) &&
    (signal4 == p_value2))
    begin
    ...
    end
```

- after -

```
if(((signal1 == 2'b10) && (signal2 != c_value1))
     ||(signal3 == 2'b10)
    )
    &&(signal4 == p_value2)
   )
    begin
    ...
    end
```

- or -

```
if  ((((signal1 == 2'b10) &&
       (signal2 != c_value1)) ||
       (signal3 == 2'b10)) &&
       (signal4 == p_value2))
    begin
    ...
    end
```

Better yet, the complexity of the above conditional should be avoided whenever possible!

- The conditional statement is allowed as long as it does not extend beyond so many lines.

```
// in case of one line conditional statement
assign wire1 = (code == 2'b01) ? (true_expression) : (false_expression);

// in case of multiple line conditional statement
assign wire1 = (code == 2'b01) ? (true_expression_1) :
               (code == 2'b10) ? (true_expression_2) :
               (code == 2'b11) ? (true_expression_3) : (false_expression);
```

6.4. RTL Mandatory

Because of synthesis reasons, there are obligatory avoidances as follows in RTL coding.

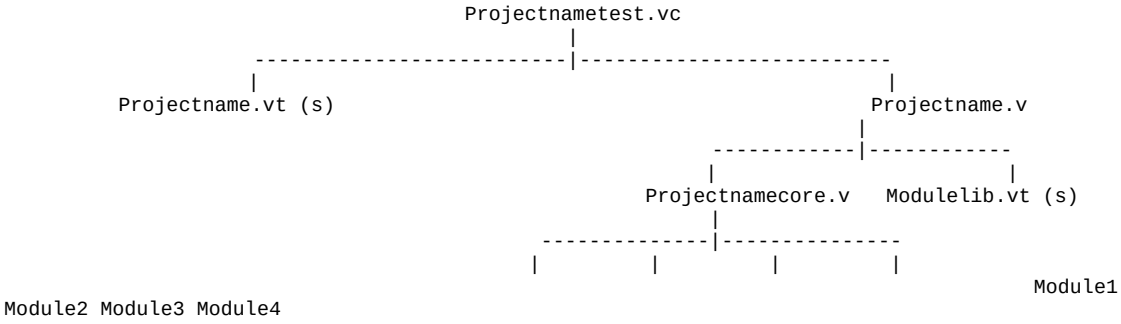
- All comment in all Verilog files has to be in English.
- Do not use the **include** statement in RTL source code. The **include** statement is replaced by forcing a simulator to search included verilog files.
- Do not use the **display** statement in RTL source code.
- Do not assign one Register in two different always blocks.
- Do not duplicate a conditional expression more than one place. For example:

```
assign cond_true   = request1 | request2 | request3;

// using cond_true in the 1st always statement
always @ (posedge clk or negedge rst_)
    begin
    if (!rst_)
        begin
        end
    else
        begin
        statement(s)
        if (cond_true) reg1 <= 2'b10;     // DO NOT duplicate like "if (request1| ..)"
```

```
            end
        end

    // using cond_true in the 2nd condition statement
        assign wire3 = cond_true ? regtrue : regfalse;       // DO NOT duplicate
                                                             // like "= (request1| ..) ?"
```

6.5. Project Hierarchy

The project hierarchy is shown as follows. The test benches can be reused in convenience. Besides, the test cases are simpler that will shorten simulation time so much.



The project directory hierarchy consists of:

```
ProjectName     +-- v         (for rtl source codes)
                +-- lib       (for simulation libraries)
                +-- vc        (for testcases)
                +-- vt        (for testbenches)
                +-- rsim      (for simulation scripts and simulation shortcuts)
```

6.6. Other Important Considerations

Empty section.

7. About This Document

The rules and recommendations presented in this document are by no means exhaustive, but should provide a solid foundation to continue development in Verilog. Suggestions for improvements and additions are welcome.